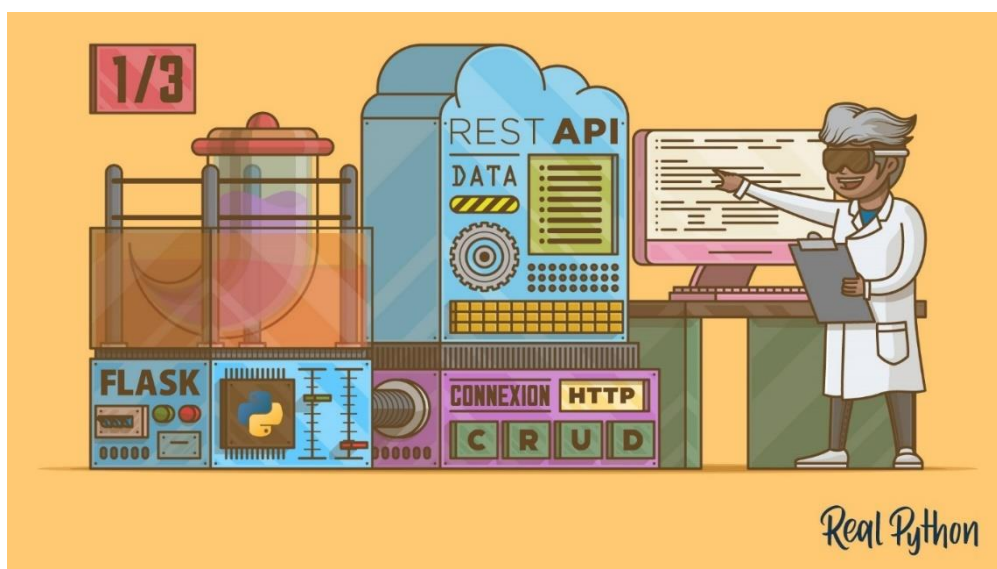


## Projet TP7 - Bloc 2 : Développement d'une application informatique

### API BACKEND SWAGGER TESTS



Formation Data Engineer – Préparation au Titre professionnel « Expert Informatique et Systèmes d'information » Option Big Data, Reconnu par l'État Niveau 7 (Niveau Bac+5) Inscrit au RNCP Code RNCP36286

Réalisé par :

Dorian BOEL - Ilyes BAHLAGUI  
Robin BRUNNEL- Zaid OTTMANI

## Table des matières

Projet TP7 - Bloc 2 : Développement d'une application informatique API BACKEND SWAGGER TESTS .	1
Table d'illustration .....	3
Cahier des charges : .....	4
Objectifs finaux pour la notation du Projet : .....	4
Conception et architecture système : .....	6
Stack technique utilisé : .....	6
conception : .....	6
Base de données : .....	7
connexion .....	7
Test création d'un client : .....	<b>Erreur ! Signet non défini.</b>
Le Swagger.....	10
Utilisation de Swagger avec Flask et Flask-RESTX.....	10
Interface graphique : .....	11
Les Tests : .....	13
Tests unitaires : .....	13
Exemple Test Client : .....	13
Exécution & Réponse : .....	13
Tests Postman : .....	14
POST : .....	14
Get : .....	14

## Table d'illustration

Figure 1 : Diagramme de classe.....	6
Figure 2: les modèles de la BDD - ex Client .....	7
Figure 3 : configuration & connexion à la BDD.....	8
Figure 4 : fichier .env - variables globales .....	9
Figure 5 : INSERT d'un client et création de la BDD.....	9
Figure 6 : capture insert de la BDD.....	9
Figure 7: Réponse API - Endpoint /clients .....	9
Figure 8 : Définition d'un namespace.....	10
Figure 9 : interface Swagger - doc API .....	11
Figure 10 : Test Get depuis la doc Swagger .....	12
Figure 11 : Test unitaire - ex Client .....	13
Figure 12 : Exécution et réponse d'un test.....	13
Figure 13 : POSTMAN - Méthode POST .....	14
Figure 14 : POSTMAN - Méthode GET .....	15

## Cahier des charges :

Livrable :

Livrables par un lien Git par groupe :

- Une branche par développeur et les branches : test, dev et main
- Sources python par projet (SOURCES PYTHON, Scripts, MySQL)
- Un dossier avec la documentation
- Un dossier contenant les scénarios de tests et les scripts
- Un dossier contenant la documentation technique (contenant l'architecture, les éléments changés selon le cahier des charges fournis, la description des serveurs virtuels python)
- Un README.md pour présenter le projet et faciliter l'utilisation
- Faire un requirements.txt pour figer les dépendances
- Suivre la prise en charge des types
  - Étudier la solution proposée par Bruno Piangerelli
- Réaliser l'API Backend en Python avec une base MySQL
  - Entités à réaliser : Client, Objet, Commande, Utilisateur
  - Optionnel : Conditionnement
  - /!\ Pas de delete physique /!\ Désactivation pour les logs. →
- Mettre en place l'API Swagger pour tester le futur Front
- Mettre en place les tests de l'API Backend avec unittest ou Pytest

## Objectifs finaux pour la notation du Projet :

L'application livrée est fonctionnelle.

Toutes les fonctionnalités prévues sont comprises dans l'application livrée.

La revue de code est pertinente et de qualité et contient des recommandations pour l'équipe de développeurs.

Les conventions d'écriture sont respectées.

Le code est lisible, structuré et sécurisé :

- Le code est commenté
- Les variables sont nommées
- Les normes du langage sont respectées
- Le code n'est pas dupliqué

La stratégie de tests comprend :

- Les niveaux de tests
- Les objectifs
- Les responsabilités
- Les tâches principales
- Les critères d'entrées et de sorties
- Les risques

## Conception et architecture système :

### Stack technique utilisé :

- Base de données relationnelle MYSQL Stocke et gère les données de l'application de manière structurée en utilisant des tables relationnelles.
- SQLAlchemy : ORM (Object-Relational Mapping), Permet de manipuler la base de données via des objets Python plutôt que des requêtes SQL directes.
- Flask : Framework web léger, facilite le développement d'applications web et d'API, offrant des outils pour gérer les routes et les requêtes HTTP.
- Flask-Marshmallow : ((Dé)Sérialisation et validation des données) il Convertit les objets Python en formats comme JSON et inversement, tout en validant les données.
- Flask-RESTX : Simplifie le développement d'API RESTful, offrant des outils pour la documentation automatique, la gestion des routes, et la validation des données.
- Unittest : Automatise les tests pour vérifier que les différentes parties du code fonctionnent comme prévu, prévenant ainsi les régressions.

### conception :

Pendant la phase de conception, nous nous sommes basés sur l'étude de cas de la fromagerie Digicheese, une proposition de solution qui avait été réalisée précédemment dans le cadre d'un projet de conception d'application informatique utilisant UML

La solution présente ce diagramme de classe :

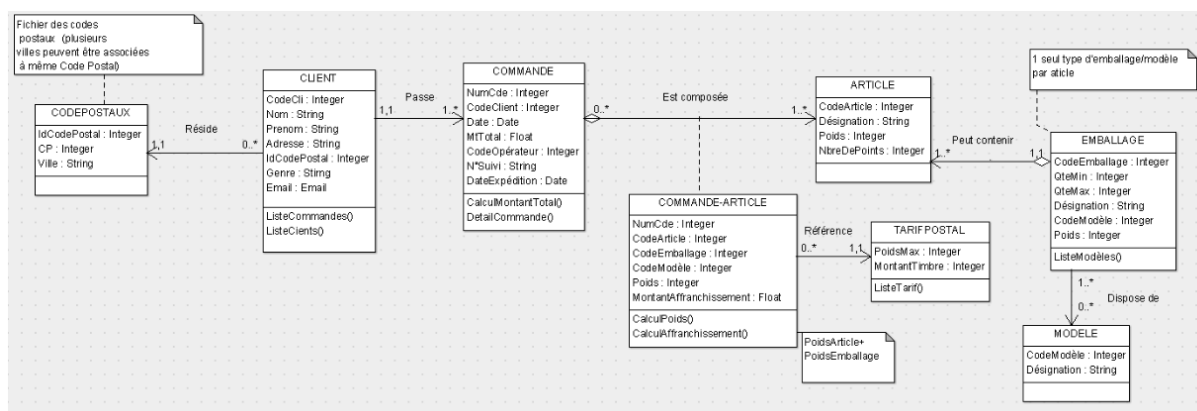


Figure 1 : Diagramme de classe

:

Pour des soucis de simplification le cahier des charges recommande de se concentrer que sur les entités suivantes : Client, Objet, Commande, Utilisateur et commande-article.

A partir de là on a défini nos cinq modèles :

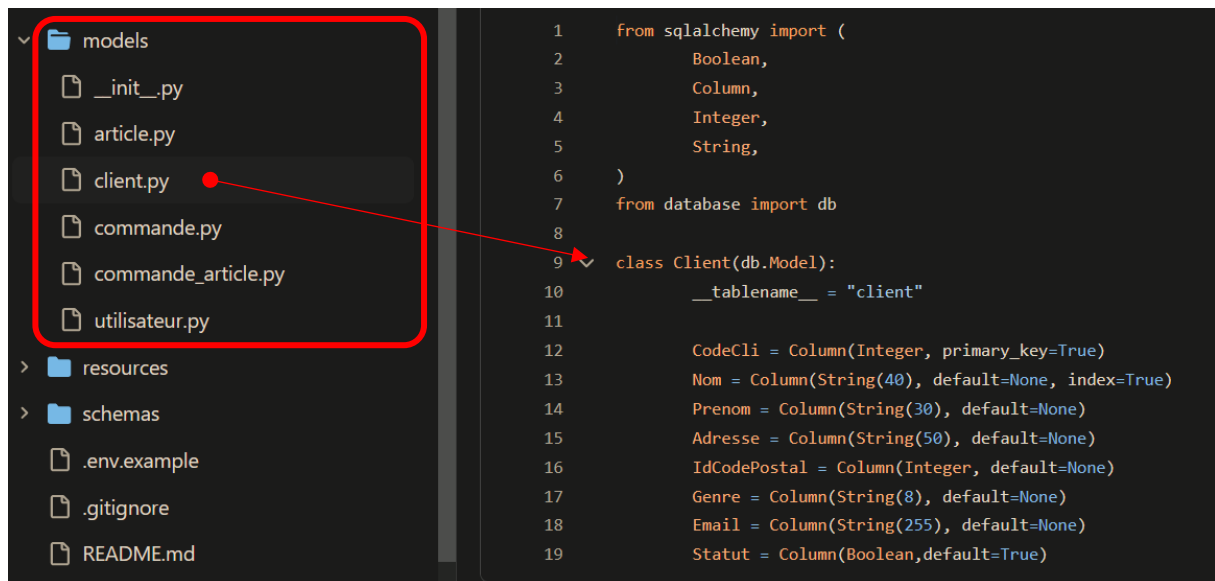


Figure 2: les modèles de la BDD - ex Client

A droite arborescence dossier, à droite le modele client.py

L'utilisation de SQLAlchemy comme ORM (Object-Relational Mapping) permet de définir des modèles de données complexes avec des relations entre les tables de manière claire et structurée, ce qui est essentiel pour un projet avec des données interconnectées (voir concept clés primaire/étrangère).

## Base de données :

### connexion

Pour se connecter à la base de données deux choses à fournir :

- Le lien : SQLALCHEMY\_DATABASE\_URI

```
1  from flask_sqlalchemy import SQLAlchemy
2  from sqlalchemy import create_engine
3  from sqlalchemy.orm import DeclarativeBase
4  from os import environ
5
6  # connexion a la base de donnée et déclaration de la base avec sql alchemy
7  user = environ.get("DB_USER")
8  password = environ.get("DB_PASSWORD")
9  port = environ.get("DB_PORT")
10 database = environ.get("DB_DATABASE_NAME")
11 host = environ.get("DB_HOST")
12
13 # url de connexion de la base
14 SQLALCHEMY_DATABASE_URI = "mysql+pymysql://{0}:{1}@{2}:{3}/{4}".format(
15     user,
16     password,
17     host,
18     port,
19     database,
20 )
```

Figure 3 : configuration & connexion à la BDD

- Le fichier .ENV :

le fichier .env permet de gérer les configurations de manière sécurisée et efficace, améliorant la maintenabilité et la sécurité de l'application.

Exemple configuration base de données :



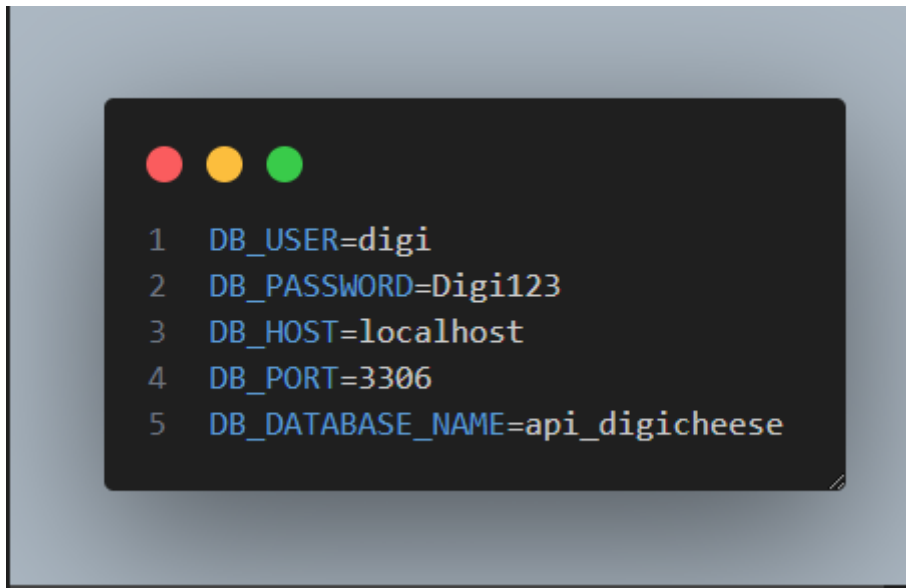


Figure 4 : fichier .env - variables globales

Ces variables sont accessibles dans l'ensemble de notre projet en important package environ : `from os import environ`.

Figure 5 : INSERT d'un client et création de la BDD

On voit bien que notre base de données reçoit l'INSERT :

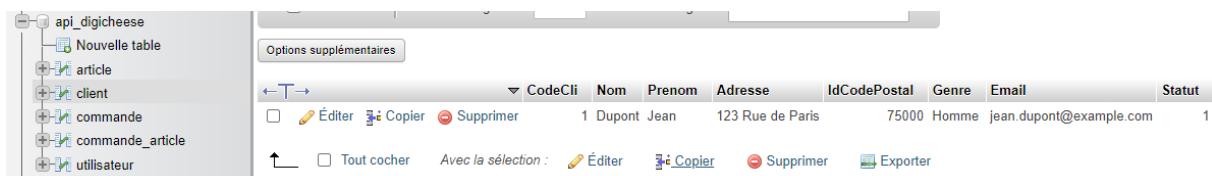


Figure 6 : capture insert de la BDD

Et que notre API répond à notre endpoint : `http://127.0.0.1:5000/clients`

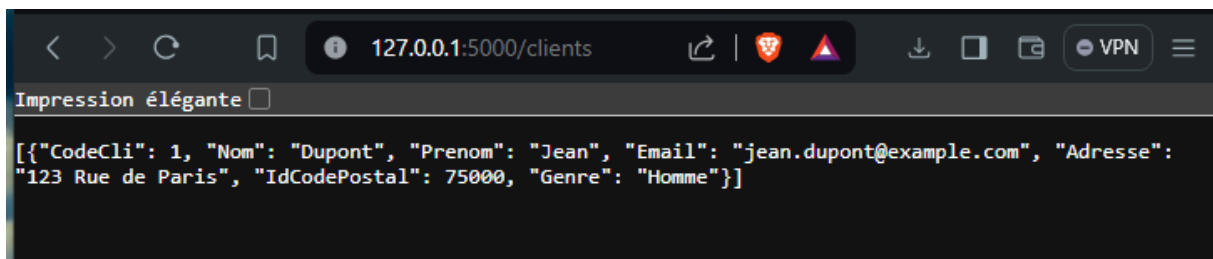


Figure 7 : Réponse API - Endpoint /clients

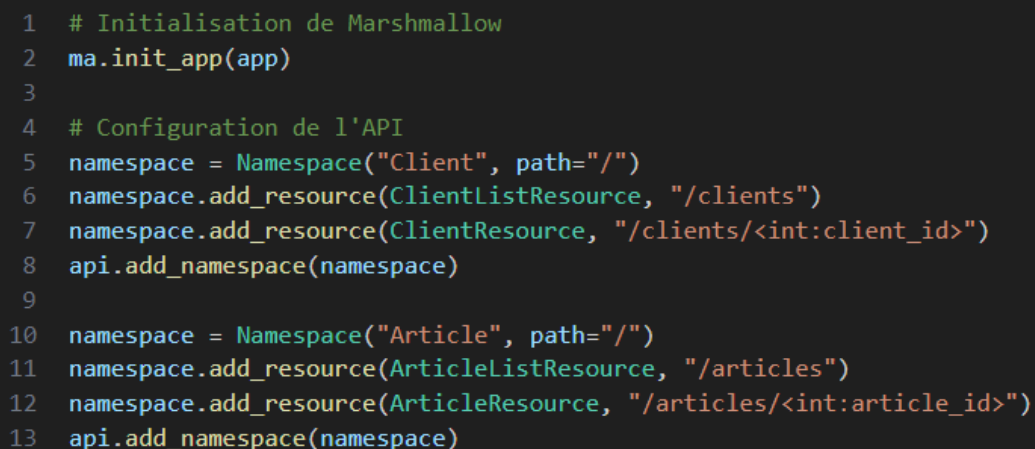
## Le Swagger

Swagger génère une documentation interactive de l'API, où les utilisateurs peuvent voir tous les Endpoints disponibles, les paramètres nécessaires, les types de réponses attendues, et tester les appels API directement depuis la documentation.

### Utilisation de Swagger avec Flask et Flask-RESTX

Flask-RESTX, une extension de Flask, intègre Swagger pour documenter automatiquement les Endpoints de l'API. Flask-RESTX est utilisé pour créer et documenter les API RESTful. Les namespaces organisent les Endpoints, et les décorateurs de ressource (`add_resource`) indiquent quelles classes gèrent les différentes routes.

Endpoint Flask :



```
1 # Initialisation de Marshmallow
2 ma.init_app(app)
3
4 # Configuration de l'API
5 namespace = Namespace("Client", path="/")
6 namespace.add_resource(ClientListResource, "/clients")
7 namespace.add_resource(ClientResource, "/clients/<int:client_id>")
8 api.add_namespace(namespace)
9
10 namespace = Namespace("Article", path="/")
11 namespace.add_resource(ArticleListResource, "/articles")
12 namespace.add_resource(ArticleResource, "/articles/<int:article_id>")
13 api.add_namespace(namespace)
```

Figure 8 : Définition d'un namespace

Un namespace dans Flask-RESTX est utilisé pour organiser les routes de l'API de manière modulaire et lisible.

`namespace = Namespace("Client", path="/")` : Crée un namespace nommé "Client" avec le chemin de base "/".

**add\_resource** : Associe des ressources (des classes qui gèrent les requêtes HTTP) à des routes spécifiques.

**namespace.add\_resource(ClientResource, "/clients/<int:client\_id>")** : Associe la ressource ClientResource à la route "/clients/[int:client\\_id](#)", permettant ainsi d'accéder à des clients spécifiques par leur identifiant.

**api.add\_namespace(namespace)** : Ajoute le namespace à l'instance de l'API, intégrant ainsi les routes définies dans le namespace à l'API globale.

Interface graphique :

Accessible depuis la racine : <http://127.0.0.1:5000/> on a cette interface :

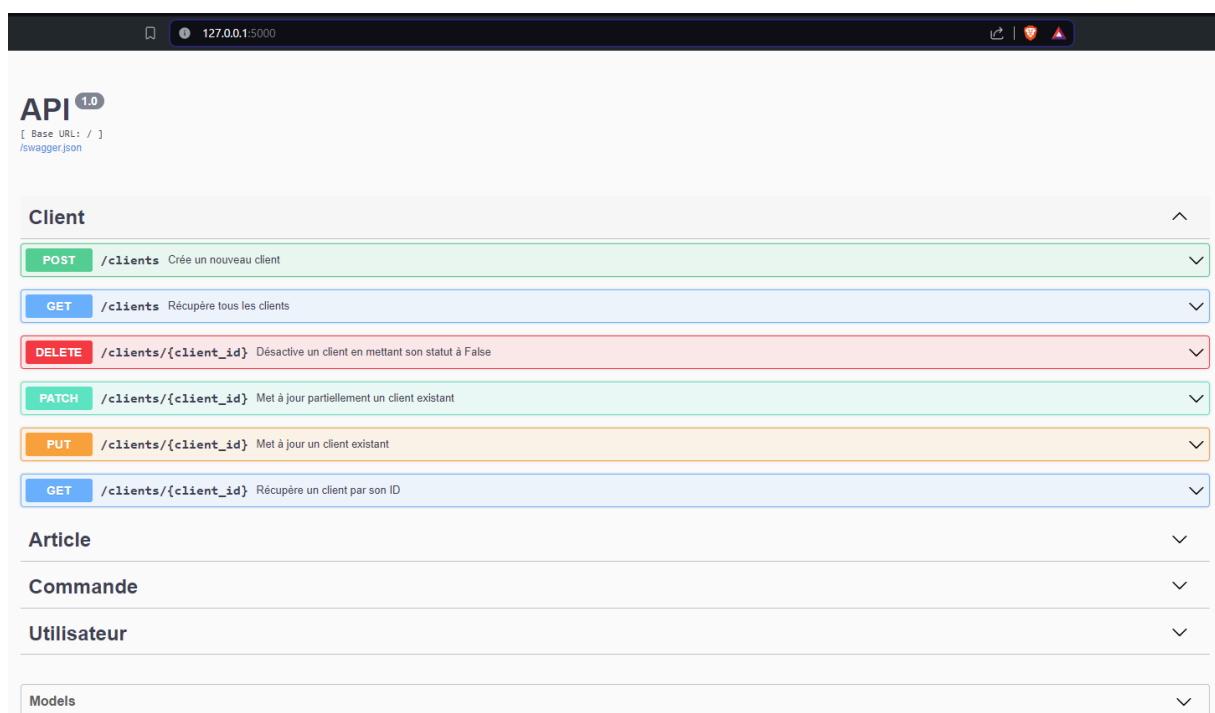


Figure 9 : interface Swagger - doc API

Test méthode GET :

Execute

Clear

Responses

Response content type application/json

Curl

curl -X 'GET' \n'http://127.0.0.1:5000/clients' \n-H 'accept: application/json'

Request URL

http://127.0.0.1:5000/clients

Server response

Code

Details

200

Response body

```
{
  "CodeCli": 1,
  "Nom": "Dupont",
  "Prenom": "Jean",
  "Email": "jean.dupont@example.com",
  "Adresse": "123 Rue de Paris",
  "IdCodePostal": 75000,
  "Genre": "Homme"
},
{
  "CodeCli": 2,
  "Nom": "zaïd",
  "Prenom": "Jean",
  "Email": "jean.dupont@example.com",
  "Adresse": "123 Rue de Paris",
  "IdCodePostal": 75000,
  "Genre": "Homme"
}
}
```

Download

Response headers

connection: close
content-length: 317
content-type: application/json
date: Thu, 06 Jun 2024 07:38:05 GMT
server: Werkzeug/3.0.3 Python/3.12.3

Responses

Code	Description
200	Success

Figure 10 : Test Get depuis la doc Swagger

## Les Tests :

### Tests unitaires :

#### Exemple Test Client :

```
1 import unittest
2 from app import app
3
4
5 class TestClients(unittest.TestCase):
6     def test_get_clients(self):
7         with app.test_client() as client:
8             response = client.get('/clients')
9             self.assertEqual(response.status_code, 200, "Status code != 200")
10            expected_data = [ { "CodeCli": 1,
11                               "Nom": "Dupont",
12                               "Prenom": "Jean",
13                               "Email": "jean.dupont@example.com",
14                               "Adresse": "123 Rue de Paris",
15                               "IdCodePostal": 75000,
16                               "Genre": "Homme" } ]
17            self.assertEqual(response.json, expected_data,
18                             "La réponse ne correspond pas aux données attendues")
19
20 if __name__ == '__main__':
21     unittest.main()
```

Figure 11 : Test unitaire - ex Client

Ce test unitaire envoie une requête à l'Endpoint `/clients` de l'application Flask, puis vérifie si la réponse et le code de statut sont conformes aux attentes.

#### Exécution & Réponse :

```
(venv)
Dell@DESKTOP-P5KFN77 MINGW64 /c/Users/Dell/Downloads/12 - PROJET - Déve[
pi/PROJETTP7_API_WEB_2024_D03 (zaid-dev)
• $ python -m unittest discover -s test -p "test_*.py"
.
-----
Ran 1 test in 0.032s
OK
```

Figure 12 : Exécution et réponse d'un test

Tests Postman :

POST :

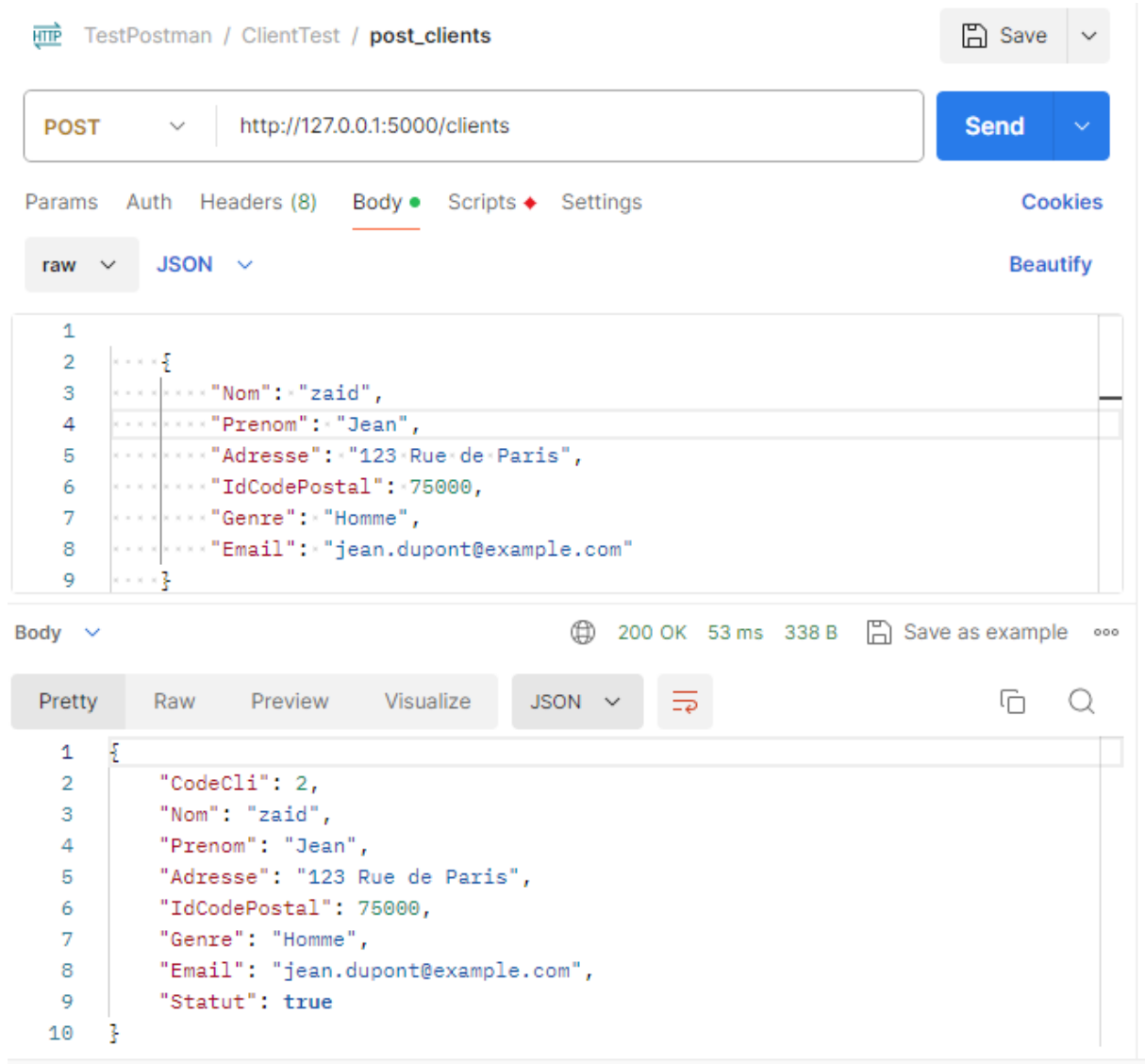


Figure 13 : POSTMAN - Méthode POST

Pour tester la méthode POST de notre API en envoi une requête à l'Endpoint Clients/ avec les données du client dans le corps de la requête (Body).

➤ Ensuite on vérifie avec la méthode GET ensuite :

Get :

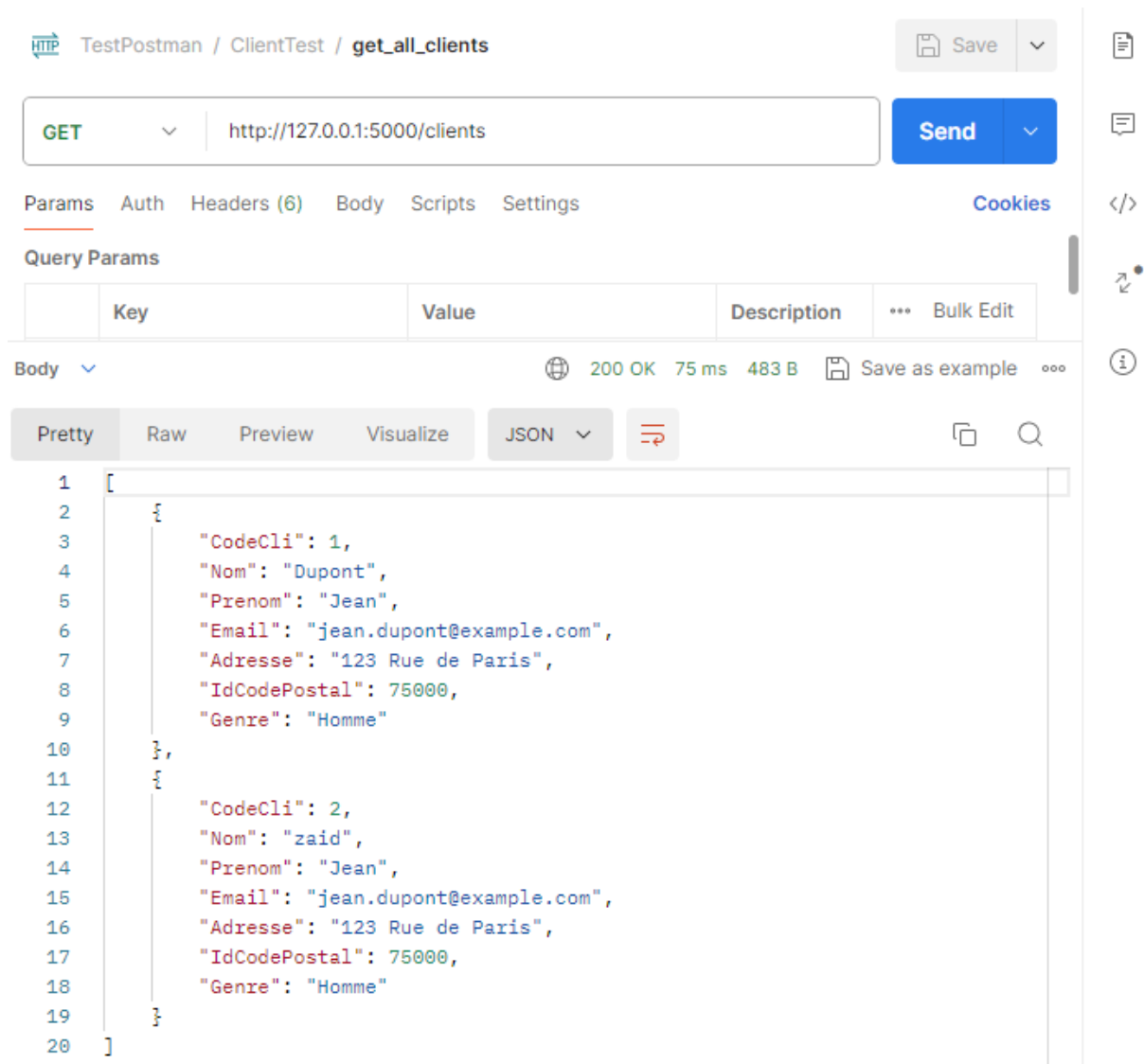


Figure 14 : POSTMAN - Méthode GET

L'API répond comme attendu.

## Conclusion :

Dans notre projet d'API Flask, nous avons appris beaucoup sur le développement d'une API, malgré quelques obstacles. Certains d'entre nous ont eu du mal à configurer leur environnement de développement, et nous avons rencontré des problèmes avec Git, notamment des conflits de Git Merge. Malgré ces difficultés, nous avons réussi à fournir un travail presque complet. Pour aller plus loin, nous devons finaliser les tests et mettre en place les relations many-to-many entre les tables de notre base de données. Nous sommes fiers de notre progrès et motivés à continuer à améliorer notre projet.